

### چکیده:

در این تمرین، به بررسی ۳ عملیات پایه روی تصاویر می پردازیم. در بخش اول، با گسسته سازی مقادیر پیکسل های تصویر و نحوه تاثیر آن بر کیفیت عکس آشنا می شویم. در بخش دوم، دو متد کاهش ابعاد تصویر، یکی حذف سطر و ستون و دیگری میانگین گیری را بررسی می کنیم. همچنین برای افزایش ابعاد تصویر نیز، دو متد کپی و تکرار و تبدیل دو خطی را پیاده سازی می کنیم. در بخش آخر نیز، به بررسی تاثیر تعداد بیت های مورد استفاده در ذخیره سازی تصویر، بر روی کیفیت عکس می پردازیم.

## ۱.۱.۱

### توضیح فنی:

در سلول اول، ابتدا به محاسبه هیستوگرام محلی برای تصویر elaine پرداخته ایم.

```
1 elaine = cv2.imread('Elaine.bmp',cv2.IMREAD_GRAYSCALE)
2
3 # equalize elaine image
4 elaine_histo = calc_histogram(elaine)
5 normal_elaine_histo = normalizeHistogram(elaine_histo,elaine.shape[0],elaine.shape[1])
6 elaine_cdf = calc_cdf(normal_elaine_histo)
7 equal_elaine = reMap(elaine,elaine_cdf)
8
```

در اینجا، ابتدا با تابع calc\_histogram، هیستوگرام تصویر را محاسبه می‌کنیم. سپس با تابع normalizationhistogram، تصویر را نرمال‌سازی می‌کنیم. در گام بعد، تابع calc\_cdf برای محاسبه cdf تصویر استفاده می‌شود. در انتها، تصویر جدید مطابق cdf محاسبه شده، توسط تابع reMap ساخته می‌شود. تابع calc\_histogram به شکل زیر تعریف می‌شود:

```
14 def calc_histogram(image):
15     length = image.shape[0]
16     width = image.shape[1]
17     pdf = np.zeros(256)
18
19     for i in range(length):
20         for j in range(width):
21             k = image[i][j]
22             pdf[k] +=1
23
24     return pdf
```

در این تابع، عکس مورد نظر به عنوان ورودی تابع استفاده می‌شود. در دو حلقه تو در تو، فراوانی رخ داد هر پیکسل که مقداری بین ۰ تا ۲۵۵ دارد را، در یک آرایه به نام pdf ذخیره می‌کنیم. طول این آرایه ۲۵۶ است، و اندیس k این آرایه، نشان دهنده فراوانی پیکسل‌هایی با مقدار k است.

برای اجرای روال متعادل‌سازی هیستوگرام، در ابتدا نیاز داریم تا طبق الگوریتم ارائه شده در درس، هیستوگرام فعلی را نرمالایز کنیم. برای اینکار یک تابع به نام normalizeHistogram نوشته ایم.

```

26 def normalizeHistogram(pdf,width,length):
27     normal_pdf = np.zeros(len(pdf))
28
29     for i in range(len(normal_pdf)):
30         normal_pdf[i] = pdf[i]/(width*length)
31
32     return normal_pdf

```

در این تابع، مقادیر pdf را به رنج بین ۰ تا ۱ مپ می کنیم. چون ماکسیمم مقدار فراوانی برابر حاصل ضرب طول در عرض تصویر است، برای تقسیم کردن در مخرج از این عدد استفاده می کنیم.

```

34 def calc_cdf(normal_pdf):
35     cdf = np.zeros(len(normal_pdf))
36
37     for i in range(len(normal_pdf)):
38         for j in range(i):
39             cdf[i] += normal_pdf[j]
40
41         cdf[i] *=255
42         cdf[i] = round(cdf[i])
43
44     return cdf

```

در این تابع، به ازای مقدار  $i$  از آرایه نرمال شده، تمام اندیس های ۰ تا  $i$  از آرایه نرمال شده را با هم جمع می کنیم. طبق فرمول محاسبه cdf، در حلقه بیرونی، که در واقع انتهای سیگمای جمع مقادیر نرمالایز شده تا آن اندیس است، مقدار آن اندیس را در ۲۵۵ ضرب می کنیم. در انتها چون این مقدار ممکن است یک عدد اعشاری باشد، آن را به مقدار صحیح نزدیک تر به آن گرد می کنیم.

```

46 def reMap(image,target_cdf):
47     newImage = image.copy()
48     for i in range(newImage.shape[0]):
49         for j in range(newImage.shape[1]):
50             newImage[i][j] = target_cdf[newImage[i][j]]
51
52     return newImage

```

این تابع با دریافت عکس هدف و cdf مورد نظر، پیکسل های عکس را با مقادیر جدید بر اساس cdf داده شده، آپدیت می کند.

در ادامه، ابتدا تصویر اصلی را با لول های مختلف، از ۲ تا ۱۲۸ لول، نمایش می دهیم.

```

1 # quantize original image
2 img2level = quantize_simulation(elaine,1) # 2 level = 2**1
3 img4level = quantize_simulation(elaine,2) # 4 level = 2**2
4 img8level = quantize_simulation(elaine,3) # 8 level = 2**3
5 img16level = quantize_simulation(elaine,4) # 16 level = 2**4
6 img32level = quantize_simulation(elaine,5) # 32 level = 2**5
7 img64level = quantize_simulation(elaine,6) # 64 level = 2**6
8 img128level = quantize_simulation(elaine,7) # 128 level = 2**7

```

این قطعه کد، پیکسل های عکس elaine را با کمک تابع quantize\_simulation، در لول های مشخص شده، تقسیم بندی می کند.

```

3 def quantize_simulation(image, n_bits):
4     coeff = 2**8 // 2**n_bits
5     return (image // coeff) * coeff
6

```

این تابع، با تقسیم ۲۵۶ بر  $2^{n\_bits}$ ، تعداد دسته های مورد نظر را محاسبه می کند. سپس پیکسل های تصویر را بر این مقدار تقسیم و به پایین گرد می کند. در نهایت حاصل را دوباره در coeff ضرب می کنیم. منطق این عملیات این است که ابتدا پیکسل ها به اعدادی بین ۰ تا level-1 مقداری که می خواهیم نرمالایز می شوند، و در ادامه دوباره در coeff ضرب می کنیم تا به اعداد ابتدای بازه ها مپ شوند.

```

1 # calculate MSE, for original image cases
2 mse2level = mean_square_error(elaine,img2level)
3 print(f'mean_square_error(2) = {mse2level}')
4 mse4level = mean_square_error(elaine,img4level)
5 print(f'mean_square_error(4) = {mse4level}')
6 mse8level = mean_square_error(elaine,img8level)
7 print(f'mean_square_error(8) = {mse8level}')
8 mse16level = mean_square_error(elaine,img16level)
9 print(f'mean_square_error(16) = {mse16level}')
10 mse32level = mean_square_error(elaine,img32level)
11 print(f'mean_square_error(32) = {mse32level}')
12 mse64level = mean_square_error(elaine,img64level)
13 print(f'mean_square_error(64) = {mse64level}')
14 mse128level = mean_square_error(elaine,img128level)
15 print(f'mean_square_error(128) = {mse128level}')

```

در نهایت در قطعه کد بالا، mean square error را بین عکس اصلی و عکس های گسسته سازی شده محاسبه می کند.

در سلول های بعد، همین روال را برای عکس متعادل شده تکرار کرده ایم.

## نتیجه:

در ابتدا، با مقایسه عکس اصلی و متعادل شده، با توجه به عکس های حاصل، تاثیر متعادل سازی در افزایش کنتراست تصویر و جدا شدن لبه های سیاه از سفید و تشخیص بهتر جزئیات را مشاهده می کنیم.

در مورد گسسته سازی (quantize)، می بینیم که پس از ۸ یا ۱۶ سطح، دیگر تفاوت قابل تشخیص با چشم نیست.

در مورد محاسبه تفاوت بین عکس ها، اعداد زیر به دست آمد:

2	4	8	16	32	64	128	Level
5436.5977	1346.5181	325.0481	77.3164	17.3819	3.4688	0.5093	<b>Without Histeq</b>
5296.0716	1318.1048	325.0791	76.2318	17.8455	3.9539	0.5847	<b>With Histeq</b>

واضح است که هرچه تعداد سطوح به ۲۵۶ نزدیکتر باشد، مقدار خطا کمتر است. در مقایسه بین حالت متعادل شده و نشده نیز، تفاوت ارور برای عکس متعادل شده با عکس های گسسته اش، فقط در تعداد سطح پایین، خطای کمتری دارد، که البته تفاوت فاحشی نیست.

## ۱.۱.۲

### توضیح فنی:

در ابتدا عکس را لود می‌کنیم.

```
1 goldhill = cv2.imread('Goldhill.bmp', cv2.IMREAD_GRAYSCALE)
2 width , length = goldhill.shape
3 plt.imshow(goldhill,cmap=plt.gray())
```

ابعاد عکس (512,512) است.

در حالت اول، عکس را با حذف سطر و ستون‌ها به صورت یکی در میان، کاهش ابعاد می‌دهیم.

```
1 # downsample an image by a factor of 2, without using averaging filter
2 # we just select rows & columns every other one
3 downsampled_without_averaging = goldhill[::2,::2]
4 plt.imshow(downsampled_without_averaging)
```

سپس، کاهش ابعاد را با استفاده از فیلتر میانگین‌گیر، انجام می‌دهیم.

```
1 # downsample an image by a factor of 2, using averaging filter
2 downsampled_using_averaging = averaging_filter(goldhill,windowSize=3,downsaplingFactor=2)
3 plt.imshow(downsampled_using_averaging)
```

در اینجا، از تابع `averaging_filter` استفاده شده است، که به شکل زیر پیاده‌سازی شده:

```

65 def averaging_filter(image,windowSize=3,downsaplingFactor=1):
66     # apply averaging filter with 'windowSize' as filter size and then down-samp1
67     newWidth = int(image.shape[0]/downsaplingFactor)
68     newLength = int(image.shape[1]/downsaplingFactor)
69     newImage = np.zeros((newWidth,newLength))
70
71     for i in range(0, image.shape[0], downsaplingFactor):
72         for j in range(0, image.shape[1], downsaplingFactor):
73             end0 = i+windowSize
74             if end0>image.shape[0]:
75                 end0 = image.shape[0]
76
77             end1 = j+windowSize
78             if end1>image.shape[1]:
79                 end1 = image.shape[1]
80
81             buffer = np.mean(image[i:end0, j:end1], axis=(0,1))
82             newImage[int(i/downsaplingFactor), int(j/downsaplingFactor)] = buffer
83
84     return newImage

```

ورودی این تابع به ترتیب، عکس ورودی مورد نظر جهت کاهش، ساینز پنجره فیلتر میانگین گیر و نرخ کاهش حجم است.

در ابتدا، یک آرایه جهت نگهداری عکس خروجی می‌سازیم. سپس در دو حلقه تو در تو، ابتدا، یک برش از عکس در ابعاد فیلتر را انتخاب می‌کنیم، سپس میانگین مقادیر پیکسل‌های این بخش را محاسبه می‌کنیم که در متغیر `buffer` ریخته می‌شود، و در انتها این مقدار را در پیکسل متناظر در آرایه عکس خروجی قرار می‌دهیم.

```

1 # upsample an image by a factor of 2, using replication method
2 figure = plt.figure(figsize=(10,10))
3
4 upsampled_with_replication_1 = replication(downsampled_without_averaging,upsamplingFactor=2)
5 figure.add_subplot(1,2,1)
6 plt.imshow(upsampled_with_replication_1)
7 plt.title('upsampled removed rows & columns')
8
9 upsampled_with_replication_2 = replication(downsampled_using_averaging,2)
10 figure.add_subplot(1,2,2)
11 plt.imshow(upsampled_with_replication_2)
12 plt.title('upsampled averaged')

```

در قطعه کد بالا، با متد `replication`، ابعاد تصاویر را دو برابر می‌کنیم. پیاده‌سازی این متد به شکل زیر است:

```

86 def replication(image,upsamplingFactor=2):
87     # upsampling, by copying rows & columns
88     newWidth = image.shape[0]*upsamplingFactor
89     newLength = image.shape[1]*upsamplingFactor
90     newImage = np.zeros((newWidth,image.shape[1]))
91
92     for i in range(image.shape[0]):
93         for k in range(upsamplingFactor):
94             newImage[(i*upsamplingFactor)+k][:] = image[i][:]
95
96     newImage2 = np.zeros((newWidth,newLength))
97
98     for j in range(newImage.shape[1]):
99         for k in range(upsamplingFactor):
100             newImage2[:,(j*upsamplingFactor)+k] = newImage[:,j]
101
102     return newImage2

```

پارامترهای این تابع، تصویر ورودی برای تغییر ابعاد و نرخ تبدیل است. در این تابع، ابتدا ستون‌های عکس اصلی را در ستون‌های عکس خروجی، کپی و تکرار می‌کنیم، و سپس همین کار را برای سطرهای تصویر، تکرار می‌کنیم. در ادامه اینبار با استفاده از متد دوخطی، افزایش ابعاد را انجام دادیم.

```

1 # upsample an image by a factor of 2, using bilinear interpolation method
2 figure = plt.figure(figsize=(10,10))
3
4 upsampled_with_bilinear_interpolation_1 = np.zeros((downsampled_without_averaging.shape[0]*2,downsampled_without_
5 upsampled_with_bilinear_interpolation1 = interpolate_bilinear(downsampled_without_averaging,upsampled_with_biline
6 figure.add_subplot(1,2,1)
7 plt.imshow(upsampled_with_bilinear_interpolation_1)
8 plt.title('bilinear_interpolation_1')
9
10 upsampled_with_bilinear_interpolation_2 = np.zeros((downsampled_using_averaging.shape[0]*2,downsampled_using_aver
11 upsampled_with_bilinear_interpolation2 = interpolate_bilinear(downsampled_using_averaging,upsampled_with_bilinear
12 figure.add_subplot(1,2,2)
13 plt.imshow(upsampled_with_bilinear_interpolation_2)
14 plt.title(['bilinear_interpolation_2'])

```

پیاده‌سازی متد `interpolate_bilinear` با کمک یک منبع خارجی، به شکل زیر است.



```

104 def interpolate_bilinear(array_in, array_out):
105     width_in = array_in.shape[0]
106     height_in = array_in.shape[1]
107     width_out = array_out.shape[0]
108     height_out = array_out.shape[1]
109
110     for i in range(height_out):
111         for j in range(width_out):
112             # Relative coordinates of the pixel in output space
113             x_out = j / width_out
114             y_out = i / height_out

```

ورودی این تابع به ترتیب، عکس ورودی و آرایه با ابعاد مدنظر برای عکس خروجی است.

```

116         # Corresponding absolute coordinates of the pixel in input space
117         x_in = (x_out * width_in)
118         y_in = (y_out * height_in)
119
120         # Nearest neighbours coordinates in input space
121         x_prev = int(np.floor(x_in))
122         x_next = x_prev + 1
123         y_prev = int(np.floor(y_in))
124         y_next = y_prev + 1

```

```

126         # Sanitize bounds - no need to check for < 0
127         x_prev = min(x_prev, width_in - 1)
128         x_next = min(x_next, width_in - 1)
129         y_prev = min(y_prev, height_in - 1)
130         y_next = min(y_next, height_in - 1)
131
132         # Distances between neighbour nodes in input space
133         Dy_next = y_next - y_in;
134         Dy_prev = 1. - Dy_next; # because next - prev = 1
135         Dx_next = x_next - x_in;
136         Dx_prev = 1. - Dx_next; # because next - prev = 1

```

```

138         # Interpolate over 3 RGB layers
139         if (len(array_out.shape) > 2):
140             for c in range(3):
141                 array_out[i][j][c] = Dy_prev * (array_in[y_next][x_prev][c] * Dx_next + array_in[y_next][x_next][c] * Dx_prev)
142                 + Dy_next * (array_in[y_prev][x_prev][c] * Dx_next + array_in[y_prev][x_next][c] * Dx_prev)
143             else: # Interpolate over 1 grayscale layer
144                 array_out[i][j] = Dy_prev * (array_in[y_next][x_prev] * Dx_next + array_in[y_next][x_next] * Dx_prev)
145                 + Dy_next * (array_in[y_prev][x_prev] * Dx_next + array_in[y_prev][x_next] * Dx_prev)
146
147     return array_out

```

در هر بخش از کد، کامنت ها، گویای عملیات ساده ای است که رخ می دهد، پس توضیح اضافه ای ندارد.

در انتها، تفاوت عکس ها را با تصویر اصلی محاسبه کرده ایم.

```

1 # calculate MSE
2 removal_down_replication_up = mean_square_error(goldhill,upsampled_with_replication_1)
3 print(f'(MSE) removal_down_replication_up: {removal_down_replication_up}')
4
5 removal_down_bilinear_up = mean_square_error(goldhill,upsampled_with_bilinear_interpolation1)
6 print(f'(MSE) removal_down_bilinear_up: {removal_down_bilinear_up}')
7
8 averaging_down_replication_up = mean_square_error(goldhill,upsampled_with_replication_2)
9 print(f'(MSE) averaging_down_replication_up: {averaging_down_replication_up}')
10
11 averaging_down_bilinear_up = mean_square_error(goldhill,upsampled_with_bilinear_interpolation2)
12 print(f'(MSE) averaging_down_bilinear_up: {averaging_down_bilinear_up}')

```

### نتیجه:

<i>Pixel Replication</i>	<i>Bilinear Interpolation</i>	
99.0850	142.4705	<i>Averaging</i>
133.0759	65.0079	<i>Remove Row&amp;Column</i>

با توجه به مقادیر خطا محاسبه شده، بهترین روش برای کاهش ابعاد و سپس افزایش مجدد ابعاد یک تصویر، استفاده از روش حذف سطر و ستون و در ادامه تبدیل دوطبقی است.

### ۱.۱.۳

#### توضیح فنی:

```
1  barbara = cv2.imread('Barbara.bmp', cv2.IMREAD_GRAYSCALE)
2
3  # plt.imshow(barbara, cmap=plt.gray())
4  img8bit = get_bit_planes(barbara.copy(), 0b11111111)
5  img7bit = get_bit_planes(barbara.copy(), 0b01111111)
6  img6bit = get_bit_planes(barbara.copy(), 0b00111111)
7  img5bit = get_bit_planes(barbara.copy(), 0b00011111)
8  img4bit = get_bit_planes(barbara.copy(), 0b00001111)
9  img3bit = get_bit_planes(barbara.copy(), 0b00000111)
10 img2bit = get_bit_planes(barbara.copy(), 0b00000011)
11 img1bit = get_bit_planes(barbara.copy(), 0b00000001)
12
```

پس از لود تصویر، با استفاده از تابع `get_bit_planes`، تصاویر را با تعداد بیت های به ترتیب ۸ تا ۱ می سازیم.

پیاده سازی تابع `get_bit_planes` به شکل زیر است:

```
7  def get_bit_planes(image, bit_planes):
8      for i in range(image.shape[0]):
9          for j in range(image.shape[1]):
10             image[i][j] = image[i][j] & bit_planes
11  image = (((image - image.min()) / (image.max() - image.min())) * 255.0).astype('uint8')
12  return image
13
```

در این تابع، رشته بیتی مدنظر را با مقدار پیکسل های تصویر، اند منطقی می گیریم تا فقط اطلاعات بیت های مدنظر، باقی بماند.

#### نتیجه:

با توجه به نتیجه عکس های خروجی، می توان گفت، با ۴ یا ۵ بیت، دیگر نمی توان اطلاعات خوبی از تصویر داشت. البته در پاسخ به سوال دوم مطرح شده، می توان گفت در بخش هایی که فرکانس تغییرات پایین تر است، یا به بیانی، با بخش های یکدست تصویر رو به رو هستیم، می توان تصویر را با تعداد بیت های کمتر، و با کیفیت قبل نشان داد، اما بخش هایی که فرکانس تغییرات زیادتر است، به سرعت تصویر نویزی و خراب می شود (حالت برفکی پیدا می کند)

```
In [ ]: import cv2
import matplotlib.pyplot as plt
from functions import *
import numpy as np
plt.style.use(plt.style.available[5])
```

1.1.1. For two cases as without and with histogram equalization (uniform histogram), display the quantized image in (4, 8, 16, 32, 64, 128) Levels and its histograms. Also, the optimum mean square error obtained for each case. Discuss and report the results for the gray Elaine image. It should be noted, you can use `rgb2gray`, `histeq` and `immse` functions for this problem.

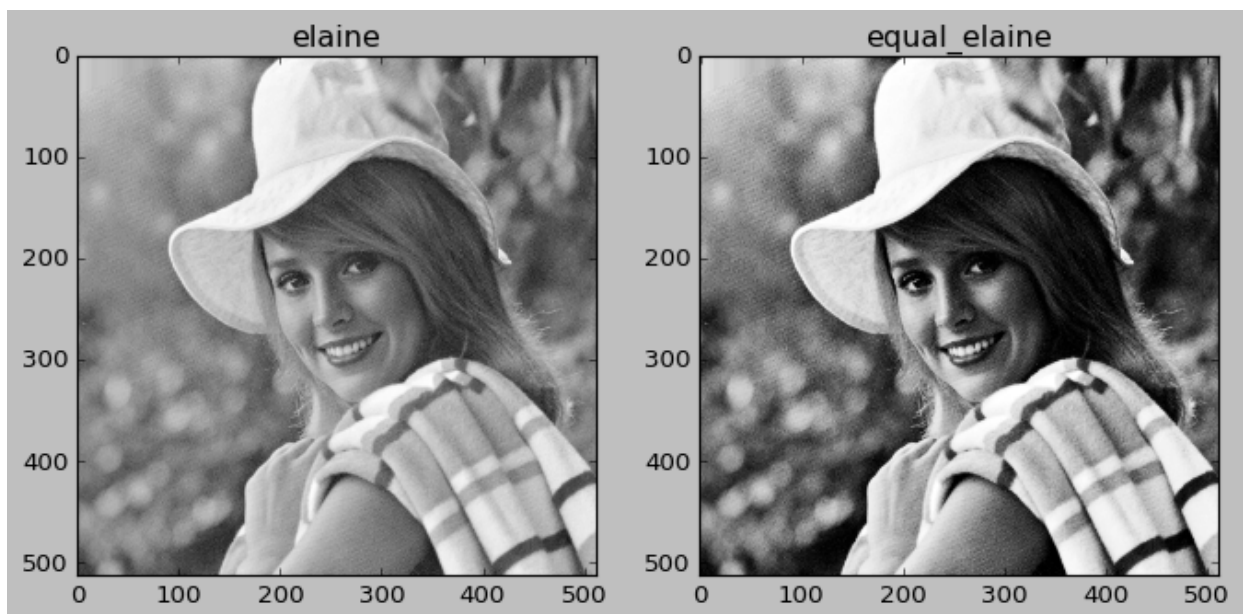
```
In [ ]: elaine = cv2.imread('Elaine.bmp', cv2.IMREAD_GRAYSCALE)

# equalize elaine image
elaine_histo = calc_histogram(elaine)
normal_elaine_histo = normalizeHistogram(elaine_histo, elaine.shape[0], elaine.shape[1])
elaine_cdf = calc_cdf(normal_elaine_histo)
equal_elaine = reMap(elaine, elaine_cdf)

# plot original and equalized image
figure = plt.figure(figsize=(10,10))
figure.add_subplot(1,2,1)
plt.imshow(elaine, cmap=plt.gray())
plt.title('elaine', color='black')

figure.add_subplot(1,2,2)
plt.imshow(equal_elaine)
plt.title('equal_elaine', color='black')
```

```
Out[ ]: Text(0.5, 1.0, 'equal_elaine')
```



```
In [ ]: # quantize original image
img2level = quantize_simulation(elaine,1) # 2 Level = 2**1
img4level = quantize_simulation(elaine,2) # 4 Level = 2**2
img8level = quantize_simulation(elaine,3) # 8 Level = 2**3
img16level = quantize_simulation(elaine,4) # 16 Level = 2**4
img32level = quantize_simulation(elaine,5) # 32 Level = 2**5
```

```
img64level = quantize_simulation(elaine,6) # 64 level = 2**6  
img128level = quantize_simulation(elaine,7) # 128 level = 2**7
```

```
In [ ]: figure = plt.figure(figsize=(12,12))  
  
figure.add_subplot(2,4,1)  
plt.imshow(elaine)  
plt.title('elaine',color='black')  
  
figure.add_subplot(2,4,2)  
plt.imshow(img2level)  
plt.title('img2level',color='black')  
  
figure.add_subplot(2,4,3)  
plt.imshow(img4level)  
plt.title('img4level',color='black')  
  
figure.add_subplot(2,4,4)  
plt.imshow(img8level)  
plt.title('img8level',color='black')  
  
figure.add_subplot(2,4,5)  
plt.imshow(img16level)  
plt.title('img16level',color='black')  
  
figure.add_subplot(2,4,6)  
plt.imshow(img32level)  
plt.title('img32level',color='black')  
  
figure.add_subplot(2,4,7)  
plt.imshow(img64level)  
plt.title('img64level',color='black')  
  
figure.add_subplot(2,4,8)  
plt.imshow(img128level)  
plt.title('img128level',color='black')  
  
plt.show()
```



```
In [ ]: # calculate MSE, for original image cases
mse2level = mean_square_error(elaine,img2level)
print(f'mean_square_error(2) = {mse2level}')
mse4level = mean_square_error(elaine,img4level)
print(f'mean_square_error(4) = {mse4level}')
mse8level = mean_square_error(elaine,img8level)
print(f'mean_square_error(8) = {mse8level}')
mse16level = mean_square_error(elaine,img16level)
print(f'mean_square_error(16) = {mse16level}')
mse32level = mean_square_error(elaine,img32level)
print(f'mean_square_error(32) = {mse32level}')
mse64level = mean_square_error(elaine,img64level)
print(f'mean_square_error(64) = {mse64level}')
mse128level = mean_square_error(elaine,img128level)
print(f'mean_square_error(128) = {mse128level}')
```

```
mean_square_error(2) = 5436.5977
mean_square_error(4) = 1346.5181
mean_square_error(8) = 325.0481
mean_square_error(16) = 77.3164
mean_square_error(32) = 17.3819
mean_square_error(64) = 3.4688
mean_square_error(128) = 0.5093
```

```
In [ ]: # quantize equalized image
img2level = quantize_simulation(equal_elaine,1) # 4 Level = 2**1
img4level = quantize_simulation(equal_elaine,2) # 4 Level = 2**2
img8level = quantize_simulation(equal_elaine,3) # 8 Level = 2**3
img16level = quantize_simulation(equal_elaine,4) # 16 Level = 2**4
```

```
img32level = quantize_simulation(equal_elaine,5) # 32 level = 2**5  
img64level = quantize_simulation(equal_elaine,6) # 64 level = 2**6  
img128level = quantize_simulation(equal_elaine,7) # 128 level = 2**7
```

```
In [ ]: figure = plt.figure(figsize=(12,12))  
  
figure.add_subplot(2,4,1)  
plt.imshow(equal_elaine,cmap=plt.gray())  
plt.title('elaine',color='black')  
  
figure.add_subplot(2,4,2)  
plt.imshow(img2level)  
plt.title('img2level',color='black')  
  
figure.add_subplot(2,4,3)  
plt.imshow(img4level)  
plt.title('img4level',color='black')  
  
figure.add_subplot(2,4,4)  
plt.imshow(img8level)  
plt.title('img8level',color='black')  
  
figure.add_subplot(2,4,5)  
plt.imshow(img16level)  
plt.title('img16level',color='black')  
  
figure.add_subplot(2,4,6)  
plt.imshow(img32level)  
plt.title('img32level',color='black')  
  
figure.add_subplot(2,4,7)  
plt.imshow(img64level)  
plt.title('img64level',color='black')  
  
figure.add_subplot(2,4,8)  
plt.imshow(img128level)  
plt.title('img128level',color='black')  
  
plt.show()
```





```
In [ ]: # calculate MSE, for equalized image case
mse2level = mean_square_error(equal_elaine,img2level)
print(f'mean_square_error(2) = {mse2level}')
mse4level = mean_square_error(equal_elaine,img4level)
print(f'mean_square_error(4) = {mse4level}')
mse8level = mean_square_error(equal_elaine,img8level)
print(f'mean_square_error(8) = {mse8level}')
mse16level = mean_square_error(equal_elaine,img16level)
print(f'mean_square_error(16) = {mse16level}')
mse32level = mean_square_error(equal_elaine,img32level)
print(f'mean_square_error(32) = {mse32level}')
mse64level = mean_square_error(equal_elaine,img64level)
print(f'mean_square_error(64) = {mse64level}')
mse128level = mean_square_error(equal_elaine,img128level)
print(f'mean_square_error(128) = {mse128level}')
```

```
mean_square_error(2) = 5296.0716
mean_square_error(4) = 1318.1048
mean_square_error(8) = 325.0791
mean_square_error(16) = 76.2318
mean_square_error(32) = 17.8455
mean_square_error(64) = 3.9539
mean_square_error(128) = 0.5847
```

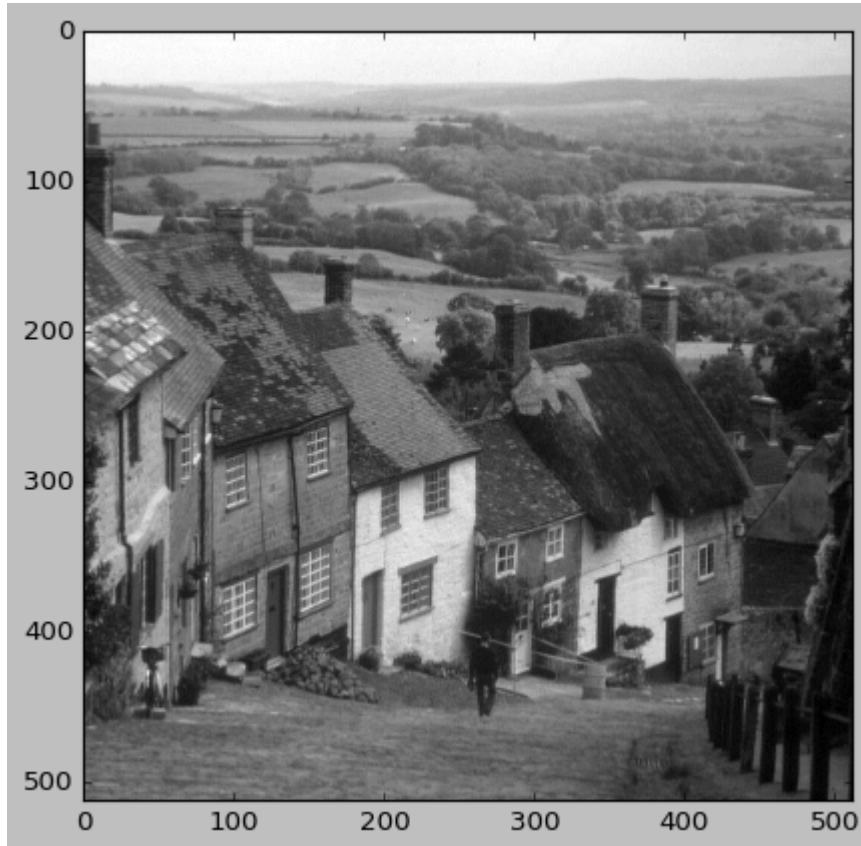
1.1.2. Write a program which can, firstly, downsample an image by a factor of 2, with and without using the averaging filter, and also, up-sample the previously downsampled images by a factor of 2, using the pixel replication and bilinear interpolation methods, respectively. Display



(zoom of image) and discuss the results obtained with different methods for the Goldhill image. Note, you can use `immse` function for this problem.

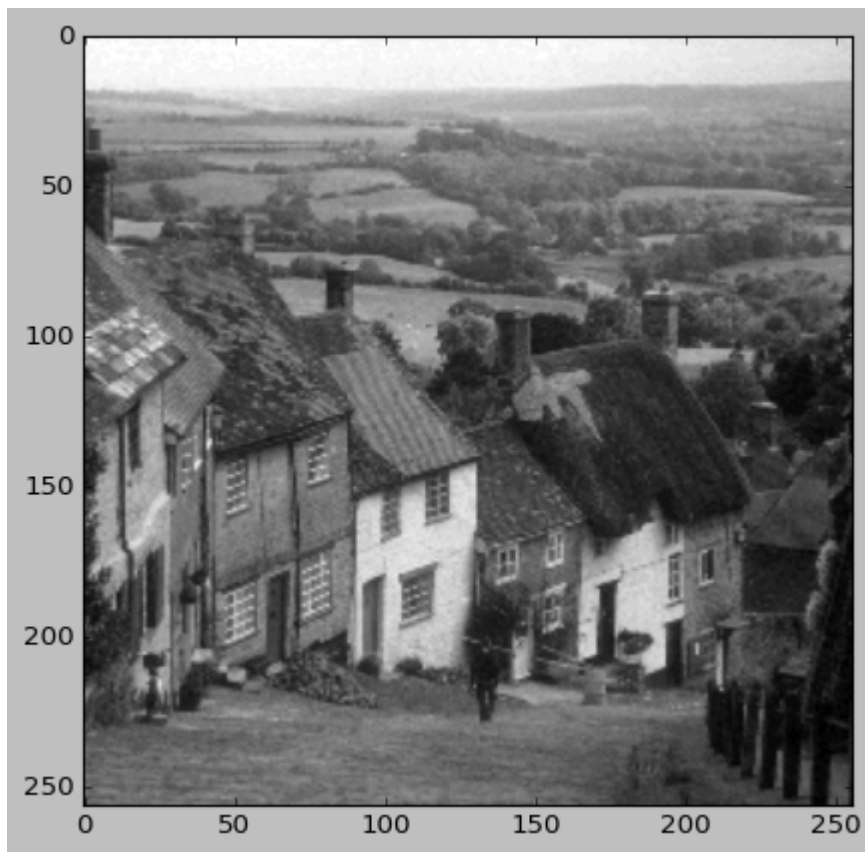
```
In [ ]: goldhill = cv2.imread('Goldhill.bmp', cv2.IMREAD_GRAYSCALE)
width , length = goldhill.shape
plt.imshow(goldhill,cmap=plt.gray())
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x2630c772dd0>
```



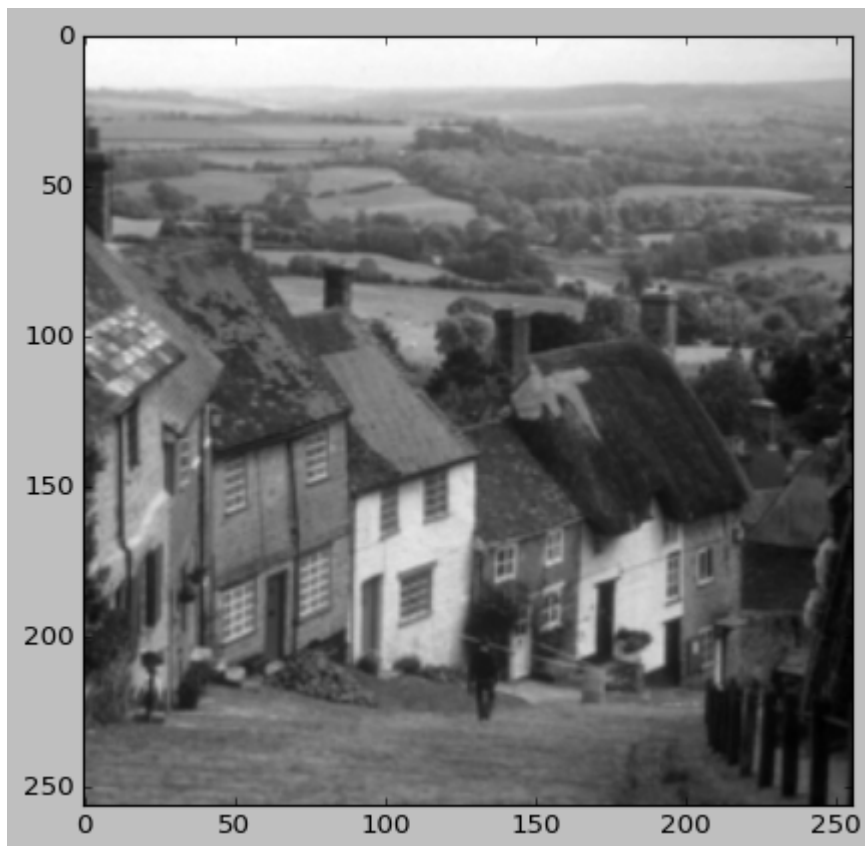
```
In [ ]: # downsample an image by a factor of 2, without using averaging filter
# we just select rows & columns every other one
downsampled_without_averaging = goldhill[::2,::2]
plt.imshow(downsampled_without_averaging)
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x2630c74bf10>
```



```
In [ ]: # downsample an image by a factor of 2, using averaging filter
downsampled_using_averaging = averaging_filter(goldhill,windowSize=3,downsaplingFactor
plt.imshow(downsampled_using_averaging)
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x2630c686c80>
```



```
In [ ]: # upsample an image by a factor of 2, using replication method
figure = plt.figure(figsize=(10,10))

upsampled_with_replication_1 = replication(downsampled_without_averaging,upsamplingFactor=2)
figure.add_subplot(1,2,1)
plt.imshow(upsampled_with_replication_1)
plt.title('upsampled removed rows & columns')

upsampled_with_replication_2 = replication(downsampled_using_averaging,2)
figure.add_subplot(1,2,2)
plt.imshow(upsampled_with_replication_2)
plt.title('upsampled averaged')

plt.show()
```

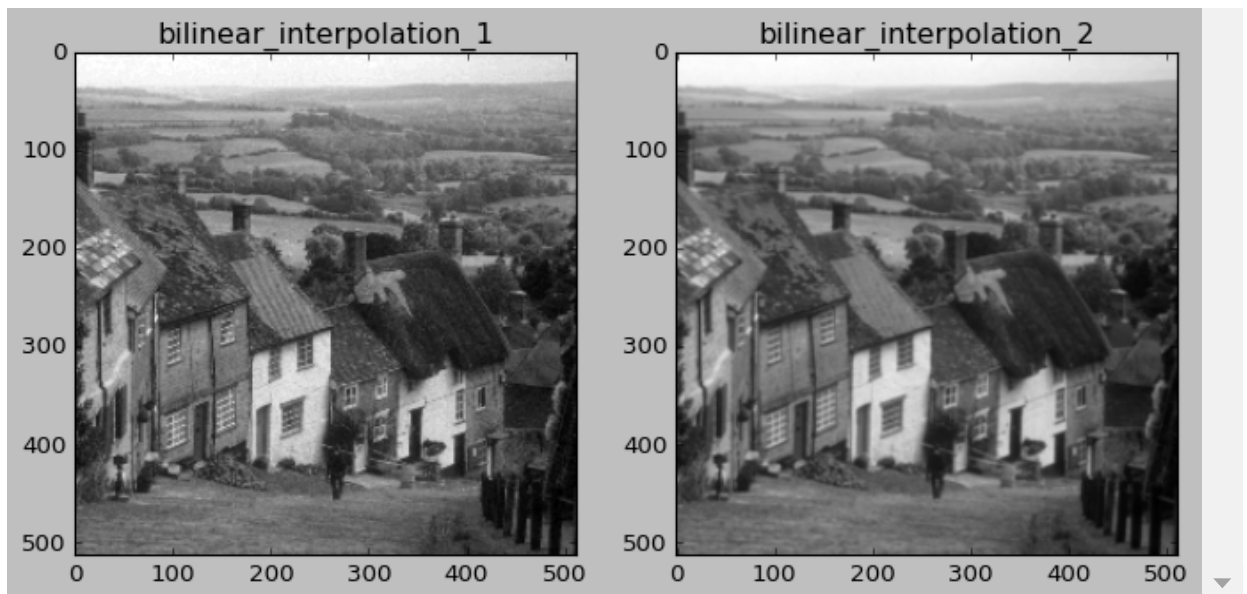


```
In [ ]: # upsample an image by a factor of 2, using bilinear interpolation method
figure = plt.figure(figsize=(10,10))

upsampled_with_bilinear_interpolation_1 = np.zeros((downsampled_without_averaging.shape[0]*2,downsampled_without_averaging.shape[1]*2))
upsampled_with_bilinear_interpolation1 = interpolate_bilinear(downsampled_without_averaging,upsampled_with_bilinear_interpolation_1)
figure.add_subplot(1,2,1)
plt.imshow(upsampled_with_bilinear_interpolation_1)
plt.title('bilinear_interpolation_1')

upsampled_with_bilinear_interpolation_2 = np.zeros((downsampled_using_averaging.shape[0]*2,downsampled_using_averaging.shape[1]*2))
upsampled_with_bilinear_interpolation2 = interpolate_bilinear(downsampled_using_averaging,upsampled_with_bilinear_interpolation_2)
figure.add_subplot(1,2,2)
plt.imshow(upsampled_with_bilinear_interpolation_2)
plt.title('bilinear_interpolation_2')
```

```
Out[ ]: Text(0.5, 1.0, 'bilinear_interpolation_2')
```



```
In [ ]: # calculate MSE
removal_down_replication_up = mean_square_error(goldhill,upsampled_with_replication_1)
print(f'(MSE) removal_down_replication_up: {removal_down_replication_up}')

removal_down_bilinear_up = mean_square_error(goldhill,upsampled_with_bilinear_interpol
print(f'(MSE) removal_down_bilinear_up: {removal_down_bilinear_up}')

averaging_down_replication_up = mean_square_error(goldhill,upsampled_with_replication_
print(f'(MSE) averaging_down_replication_up: {averaging_down_replication_up}')

averaging_down_bilinear_up = mean_square_error(goldhill,upsampled_with_bilinear_interpol
print(f'(MSE) averaging_down_bilinear_up: {averaging_down_bilinear_up}')

(MSE) removal_down_replication_up: 133.0759
(MSE) removal_down_bilinear_up: 65.0079
(MSE) averaging_down_replication_up: 99.0850
(MSE) averaging_down_bilinear_up: 142.4705
```

1.1.3. The initial image consists of eight bits of data for each pixel. Create new images using 5, 4, 3, 2 and 1 bit only for each pixel. How many bits are needed to preserve image quality? Does it change from place to place in the image? Discuss about the results. (Test Image Grayscale of Barbara).

```
In [ ]: barbara = cv2.imread('Barbara.bmp', cv2.IMREAD_GRAYSCALE)

# plt.imshow(barbara,cmap=plt.gray())
img8bit = get_bit_planes(barbara.copy(), 0b11111111)
img7bit = get_bit_planes(barbara.copy(), 0b01111111)
img6bit = get_bit_planes(barbara.copy(), 0b00111111)
img5bit = get_bit_planes(barbara.copy(), 0b00011111)
img4bit = get_bit_planes(barbara.copy(), 0b00001111)
img3bit = get_bit_planes(barbara.copy(), 0b00000111)
img2bit = get_bit_planes(barbara.copy(), 0b00000011)
img1bit = get_bit_planes(barbara.copy(), 0b00000001)

figure = plt.figure(figsize=(12,12))
figure.add_subplot(3,3,1)
plt.imshow(barbara,cmap=plt.gray())
```

```
plt.title('barbara',color='black')

figure.add_subplot(3,3,2)
plt.imshow(img8bit)
plt.title('img8bit',color='black')

figure.add_subplot(3,3,3)
plt.imshow(img7bit)
plt.title('img7bit',color='black')

figure.add_subplot(3,3,4)
plt.imshow(img6bit)
plt.title('img6bit',color='black')

figure.add_subplot(3,3,5)
plt.imshow(img5bit)
plt.title('img5bit',color='black')

figure.add_subplot(3,3,6)
plt.imshow(img6bit)
plt.title('img4bit',color='black')

figure.add_subplot(3,3,7)
plt.imshow(img3bit)
plt.title('img3bit',color='black')

figure.add_subplot(3,3,8)
plt.imshow(img2bit)
plt.title('img2bit',color='black')

figure.add_subplot(3,3,9)
plt.imshow(img1bit)
plt.title('img1bit',color='black')

plt.show()
```

